МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по практической работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Рекурсия

Студент гр. 8304		Бочаров Ф.Д.
Преподаватель		Фирсов М.А.

Санкт-Петербург
2019

Цель работы

Научиться работать с бинарным деревом, реализовав его через динамическую память.

Задание

Для заданного бинарного дерева b типа BT с произвольным типом элементов:

- *определить максимальную глубину дерева b, т. е. число ветвей в самом длинном из путей от корня дерева до листьев;*
- *вычислить длину внутреннего пути дерева b, т. е. сумму по всем узлам длин путей от корня до узла;*
- *напечатать элементы из всех листьев дерева b;*
- *подсчитать число узлов на заданном уровне n дерева b (корень считатькорень считать узлом 1-го уровня);*

Вариант 2-д.

Описание алгоритма

Подзадачи решались с помощью итераторов, которые возвращали элементы по пути обхода в глубину или в ширину.

TreeIterDFS – класс, которые помимо унаследованных функций содержит такие поля, как unsigned int size – размер элементов массива, который заполняется по пути обхода в глубину в нужном порядке.

Tree – класс, содержащий в себе такие поля, как указатель на структуру Node, которая в свою очередь содержит поля

int data,

Node*left

Node*right

Считать данные узла и указатели на его левую и правую ветку соответственно, функцию void insert и её функцию helper, нужные для рекурсивной вставки элемента в дерево, std::map<Node*, int> get_nodes_and_levels - функция возвращающая словарь, состоящий из ссылки на элемент и номера его уровня, функции TreeIterator* make_iterator_DFSи TreeIter* make_iterator_BFSвозвращают новый объект соответствующего типа.

Выполнение работы

Для реализации поставленной задачи были реализованы следующие методы в классе Tree: int dip- последовательно проходим итератором в ширину по всем элементам и вызывая функцию, которая по ссылке на узел выдаёт уровень на котором он лежит и сравниваем его с текущим, если он меньше, то переприсваиваем его и в итоге возвращаем максимум, void print_leaves- также итератором проходим по дереву и если слева и справа нет потомков, выводим этот элемент, int tree_length - вычисляет полную глубину дерева, как сумму количества элементов на данном уровне умноженную на, int count_nodes_in_level – проходит по итератору в ширину и увеличивает значение счётчика, если переданный в функцию параметр равен значению по нынешнему ключу.

Тестирование

	Входные данные	Выходные данные
	10 422 412 65 7800 20 35	Leaf: 10 Leaf: 20 Leaf: 35 Number of elements in the n level: 2 Dip of the tree: 13
2.	500 300 150 70 45 0 -9 -128 -939	Dip: 9 Leaf: -939 Number of elements in the n level: 1 Dip of the tree: 36
3.	1	Dip: 1 Leaf: 1 Number of elements in the n level: 0 Dip of the tree: 0

Выводы

В ходе выполнения работы я ознакомился с основными понятиями и приёмами рекурсивного программирования, также были получены навыки программирования рекурсивных процедур и функций на языке программирования C++ и создания бинарного дерева, реализовав его с помощью динамической памяти. Научились работать и создавать итераторы для различного типа обходов дерева.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Main.cpp

```
#include <iostream>
#include "Tree.h"

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "You must set value" << std::endl;
        return 1;
    } else {
        Tree tr(atoi(argv[1]));
        for (int i = 2; i < argc; i++) {
            tr.insert_elem(atoi(argv[i]));
        }
        tr.cout_tree();
        tr.cout_leaves();
        std::cout << tr.dip_tree() << std::endl;
        std::cout << tr.nodes_at_level(1) << std::endl;
        std::cout << tr.dip_of_tree() << std::endl;
        return 0;
    }
}
```

Iter.h

```
#pragma once
#include <vector>
#include "Tree.h"
class Node_of_tree;

class TreeIter {
public:
    virtual bool has_next() = 0;
    virtual std::shared_ptr<Node_of_tree> next() = 0;
};

class TreeIterBFS : public TreeIter {
private:
    unsigned int size = 0;
    std::map<std::shared_ptr<Node_of_tree>, bool> visiting;
    std::vector<std::shared_ptr<Node_of_tree>> breadth_path;
public:
    TreeIterBFS(const std::shared_ptr<Node_of_tree>& root);

    bool has_next();
    std::shared_ptr<Node_of_tree> next();
};

class TreeIterDFS : public TreeIter {
private:
    unsigned int size = 0;
    std::vector<std::shared_ptr<Node_of_tree>> dip_path;
    void go_in_dip(const std::shared_ptr<Node_of_tree>& a);
public:
    TreeIterDFS(const std::shared_ptr<Node_of_tree>& root);

    bool has_next();
    std::shared_ptr<Node_of_tree> next();};
```

Iter.cpp

```
#include <queue>
#include "Iter.h"
#include "Tree.h"

TreeIterDFS::TreeIterDFS(const std::shared_ptr<Node_of_tree>& root) {
    if (root != nullptr) {
        dip_path.push_back(root);
        go_in_dip(root->left_Node);
        go_in_dip(root->right_Node);
    }
}

void TreeIterDFS::go_in_dip(const std::shared_ptr<Node_of_tree>& parent) {
    if (parent == nullptr) {
        return;
    } else {
        dip_path.push_back(parent);
        go_in_dip(parent->left_Node);
        go_in_dip(parent->right_Node);
    }
}

bool TreeIterDFS::has_next() {
    if (size + 1 <= dip_path.size()){
        size++;
        return true;
    }
    return false;
}

std::shared_ptr<Node_of_tree> TreeIterDFS::next() {
    if (dip_path.size() > size)
        return dip_path[size];
    return std::shared_ptr<Node_of_tree>();
}

TreeIterBFS::TreeIterBFS(const std::shared_ptr<Node_of_tree>& root) {
    if (root != nullptr) {
        std::queue<std::shared_ptr<Node_of_tree>> queue;
        queue.push(root);
        visiting[root] = true;
        while(!queue.empty()) {
            std::shared_ptr<Node_of_tree> parent = queue.front();
            queue.pop();
            breadth_path.push_back(parent);
            if (parent->left_Node != nullptr && !visiting[parent-
>left_Node]) {
                queue.push(parent->left_Node);
                visiting[parent->left_Node] = true;
            }
            if (parent->right_Node != nullptr && !visiting[parent-
>right_Node]) {
                queue.push(parent->right_Node);
                visiting[parent->right_Node] = true;
            }
        }
    }
}

bool TreeIterBFS::has_next() {
```

```
    if (size + 1 <= breadth_path.size()){
        size++;
        return true;
    }
    return false;
}

std::shared_ptr<Node_of_tree> TreeIterBFS::next() {
    if (breadth_path.size() > size)
        return breadth_path[size];
    return std::shared_ptr<Node_of_tree>();
}
```

Tree.h

```
#pragma once
#include <memory>
#include <map>
#include <iostream>
#include <queue>
#include "Iter.h"

class TreeIter;

struct Node_of_tree{
    int value;
    std::shared_ptr<Node_of_tree> left_Node = nullptr;
    std::shared_ptr<Node_of_tree> right_Node = nullptr;
};

class Tree{
private:
    std::shared_ptr<Node_of_tree> Tree_root;
    void insert_elem(std::shared_ptr<Node_of_tree>&, int element);
    std::map<std::shared_ptr<Node_of_tree>, int> nodes_at_levels();
public:
    explicit Tree(int);

    std::shared_ptr<TreeIter> make_DFS_iter();
    std::shared_ptr<TreeIter> make_BFS_iter();

    void insert_elem(int element);
    void cout_tree();

    int dip_tree();
    void cout_leaves();
    int dip_of_tree();
    int nodes_at_level(int);
};
```

Tree.cpp

```
#include "Tree.h"
#include "Iter.h"

Tree::Tree(int root_data) {
    Tree_root = std::make_shared<Node_of_tree>();
    Tree_root->value = root_data;
}

std::shared_ptr<TreeIter> Tree::make_DFS_iter() {
    return std::make_shared<TreeIterDFS>(Tree_root);
}

std::shared_ptr<TreeIter> Tree::make_BFS_iter() {
    return std::make_shared<TreeIterBFS>(Tree_root);
}

void Tree::insert_elem(int element) {
    if (Tree_root == nullptr) {
        Tree_root = std::make_shared<Node_of_tree>();
        Tree_root->value = element;
    } else {
        if (Tree_root->value > element) {
            insert_elem(Tree_root->left_Node, element);
        }

        if (Tree_root->value < element) {
            insert_elem(Tree_root->right_Node, element);
        }

        if (Tree_root->value == element) {
            Tree_root->value = element;
        }
    }
}

void Tree::insert_elem(std::shared_ptr<Node_of_tree>& parent, int element)
{
    if (parent == nullptr) {
        parent = std::make_shared<Node_of_tree>();
        parent->value = element;
    } else {
        if (parent->value > element) {
            insert_elem(parent->left_Node, element);
        }

        if (parent->value < element) {
            insert_elem(parent->right_Node, element);
        }

        if (parent->value == element) {
            parent->value = element;
        }
    }
}

void Tree::cout_tree() {
    std::map<std::shared_ptr<Node_of_tree>, int> nodes_and_levels =
nodes_at_levels();
    std::shared_ptr<TreeIter> bfs = this->make_BFS_iter();
    int level_of_dip = 0;
```

```
    for (std::shared_ptr<Node_of_tree> el = bfs->next(); bfs->has_next();
el = bfs->next()) {
        level_of_dip = nodes_and_levels[el];
        std::cout << std::string(level_of_dip * 5, ' ') << el->value <<
std::endl;
    }
}

int Tree::dip_tree() {
    if (Tree_root == nullptr) {
        return 0;
    } else {
        std::map<std::shared_ptr<Node_of_tree>, int> nodes_and_levels =
nodes_at_levels();
        if (nodes_and_levels.size() == 1)
            return 1;
        std::shared_ptr<TreeIter> bfs = this->make_BFS_iter();
        int max_level_of_dip = 0;

        for (std::shared_ptr<Node_of_tree> el = bfs->next(); bfs-
>has_next(); el = bfs->next()) {
            if (max_level_of_dip < nodes_and_levels[el]) {
                max_level_of_dip = nodes_and_levels[el];
            }
        }

        return max_level_of_dip;
    }
}

void Tree::cout_leaves() {
    std::map<std::shared_ptr<Node_of_tree>, int> nodes_and_levels =
nodes_at_levels();
    if (nodes_and_levels.size() == 1) {
        std::cout << "Leaf: " << Tree_root->value << std::endl;
        return;
    }
    std::shared_ptr<TreeIter> dfs = this->make_DFS_iter();
    for (std::shared_ptr<Node_of_tree> el = dfs->next(); dfs->has_next();
el = dfs->next()) {
        if (el->left_Node == nullptr && el->right_Node == nullptr) {
            std::cout << "Leaf: " << el->value << std::endl;
        }
    }
}

int Tree::dip_of_tree() {
    std::map<std::shared_ptr<Node_of_tree>, int> nodes_and_levels =
nodes_at_levels();
    if (nodes_and_levels.size() == 1)
        return 1;
    std::shared_ptr<TreeIter> bfs = this->make_BFS_iter();
    int max_level = 1;
    int level_of_dip;

    for (std::shared_ptr<Node_of_tree> el = bfs->next(); bfs->has_next();
el = bfs->next()) {
        level_of_dip = nodes_and_levels[el];
        if (max_level < level_of_dip) {
            max_level = level_of_dip;
```

```
        }
    }

    int length = 0;
    for (int i = 1; i <= max_level; i++) {
        length += nodes_at_level(i) * (i - 1);
    }

    return length;
}

int Tree::nodes_at_level(int data) {
    if (Tree_root == nullptr) {
        return 0;
    } else {
        std::map<std::shared_ptr<Node_of_tree>, int> nodes_and_levels =
nodes_at_levels();
        if (nodes_and_levels.size() == 1) {
            return 1;
        }
        std::shared_ptr<TreeIter> bfs = this->make_BFS_iter();
        int level_of_dip;
        int count = 0;

        for (std::shared_ptr<Node_of_tree> el = bfs->next(); bfs-
>has_next(); el = bfs->next()) {
            level_of_dip = nodes_and_levels[el];
            if (level_of_dip == data) {
                count++;
            }
        }

        return count;
    }
}

std::map<std::shared_ptr<Node_of_tree>, int> Tree::nodes_at_levels() {
    std::map<std::shared_ptr<Node_of_tree>, int> nodes_and_levels;
    if (Tree_root == nullptr) {
        std::cout << "Empty" << std::endl;
        return std::map<std::shared_ptr<Node_of_tree>, int>();
    } else {
        std::map<std::shared_ptr<Node_of_tree>, bool> is_visit;
        std::queue<std::pair<std::shared_ptr<Node_of_tree>, int>> queue;
        queue.push({Tree_root, 1});
        is_visit[Tree_root] = true;

        while (!queue.empty()) {
            auto [parent, level] = queue.front();
            nodes_and_levels[parent] = level;
            queue.pop();

            if (parent->left_Node != nullptr && !is_visit[parent-
>left_Node]) {
                queue.push({parent->left_Node, level + 1});
                is_visit[parent->left_Node] = true;
            }

            if (parent->right_Node != nullptr && !is_visit[parent-
>right_Node]) {
                queue.push({parent->right_Node, level + 1});
```

```
                is_visit[parent->right_Node] = true;
            }
        }

        return nodes_and_levels;
    }
}
```